toString and Private Fields

toString turns a memory code into a readable result log. private plus a getter turns an open field into a door you control -- the exact shape of every Page Object.

Automation Foundations · ashishautomationlabs.store

Try to print an object directly and Java, not knowing what matters to you, prints the type name and a memory code -- something like `TestResult@1b6d3586`, which helps no one. You fix that by giving the class a `toString` method that returns the text you want:

```
class TestResult {
    String name;
    String status;
    int durationMs;

    TestResult(String name, String status, int durationMs) {
        this.name = name;
        this.status = status;
        this.durationMs = durationMs;
    }

    public String toString() {
        return name + " [" + status + ", " + durationMs + "ms]";
    }
}
```

Now the same list of objects from Chapter 5, printed, reads like a report:

```
List<TestResult> suite = new ArrayList<>();
suite.add(new TestResult("LoginTest", "PASS", 240));
suite.add(new TestResult("SearchTest", "FAIL", 512));
suite.add(new TestResult("CartTest", "PASS", 180));
for (TestResult r : suite) {
    System.out.println(r);
}

LoginTest [PASS, 240ms]
SearchTest [FAIL, 512ms]
CartTest [PASS, 180ms]
```

Whenever Java needs the text form of your object -- in `println`, in a joined `String`, in a printed list -- it calls your `toString`. This one method turns a wall of memory codes into a readable result log.

Controlling access: private fields

So far any code can reach in and change a field: `r.status = "whatever"`. Sometimes that's fine, but often you want to protect an object's data so it can only be read, or only be set through a check. You do that by marking the field `private` and offering a method to read it -- a **getter**:

```
class PageObject {
    private String url;

    PageObject(String url) {
        this.url = url;
    }

    String getUrl() {
        return url;
    }
}

PageObject login = new PageObject("https://uat.shopcart.internal/login");
System.out.println("Login page URL: " + login.getUrl());

Login page URL: https://uat.shopcart.internal/login
```

`private` means "only code inside this class may touch this field directly." Outside code must go through `getUrl()`. Right now the getter just hands the value back, but the point is control: later you could add a check, a log, or a default, in one place, and nothing outside would need to change.

This pattern -- private data, public methods -- is the reason the Page Object classes in every real Selenium framework look the way they do. You're already reading their shape: private fields for the page's elements, methods for the actions a user can take, a constructor that receives the driver. A test-data model -- a `User`, an `Order`, a `Product` -- is a class with fields and a constructor, often built straight from a row of a data file. You're not learning toward real automation code; you're learning the exact material it's made of.

> *Why make a field private if you just add a getter that returns it? Because the getter is a door you control and the field is not. Today the getter only returns the value. The day you need to log every read, validate a set, or compute the value instead of storing it, you change one method and no outside code breaks. Private fields keep that freedom.*